PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

BAKALÁŘSKÁ PRÁCE

Multi-agent systems simulator in Common Lisp

2014

Peter Vaňušanik

Anotace

Cieľ tejto bakalárskej práce je využiť jazyk Common Lisp k naprogramovaniu obecného simulátora multiagentných systémov, ktorý by bol podobný teoretickému modelu. Takto napísaný framework by mal byť schopný exportu simulácie do súboru a aj pohodlného užívateľského rozhrania.

Děkuji doc.RNDr. Michalu Krupkovi, Ph.D. za cenné rady, ochotu a trpělivost.
I would also like to thank Derrick Lam for helping with language proficiency and proofreading.

Obsah

Listings

Seznam obrázků

1. Introduction

The reason why I picked multi-agent simulators as my thesis topic is my love for the Common Lisp language. Unfortunately in modern days, Common Lisp is regarded as an old, ancient language that only academia or very old programmers use, which is simply not true. Common Lisp is as modern as any other language. In fact, many features of Common Lisp are regularly borrowed and "reinvented"as new ones for other languages, such as Java or Python.

With the help of Common Lisp, I will present a framework called `cl-ass` - **Common Lisp (multi)agent systems simulator** created for this thesis. In chapter 2, I will present a theoretical background into artificial intelligence and, more specifically, into multi agent systems. In chapter 3, I will present the Common Lisp language and its main features, along with the LispWorks IDE I have chosen to use to develop `cl-ass` in. Chapter 4 is devoted to how `cl-ass` was created and the features of Common Lisp used in its creation. Lastly, chapter 5 contains a user manual with complete API for both the framework and graphical add-on `capi-loader`.

2. Artificial Intelligence

The text in this section of the thesis is based on "Artificial Intelligence: A Modern Approach" by Stuart Russel and Peter Norvig[1]. It provides a theoretical background on artificial intelligence research and, more specifically, multi-agent systems.

2.1. What is an AI? Types of AI research

Artificial Intelligence has been around since the beginning of computer science in general. Scientists were always interested in AI, and it is one of the most exciting fields in computer science research. Before we can study AI, we must define what AI actually is. There are two main branches of AI, each with two subcategories based on their aims.

Some researchers base AI on human intelligence, and believe that true AI should be as close to human intelligence as possible. Other researchers, however, base their research on ideal intelligence, which is not based on human intelligence at all and is instead based on the concept of **rationality**.

Both of these branches also distinguish between AI that can think and AI that can act. While the two are related to each other, they are conceptually different.

We can summarize the four AI approaches along the two dimensions as:

- **Acting humanly** - The Turing test approach
- **Thinking humanly** - The cognitive modeling approach
- **Thinking rationally** - The "laws of thought" approach
- **Acting rationally** - The rational agent approach

2.1.1. Acting humanly

This approach centers around the so-called **Turing test** proposed by Alan Turing. The test is based on the idea that if a human operator is unable to clearly distinguish between another human and an AI solely based on written questions and answers, we can say the that AI is sufficiently human. **The Complete Turing test** is similar to the standard Turing test except the AI would also have to operate mechanical parts that would give and take objects from a human operator.

Required skills for the AI include: *natural language processing*, *knowledge representation*, *automated reasoning*, *machine learning*. To pass the complete test, these skills are also required: *computer vision*, *robotics*.

2.1.2. Thinking humanly

Before we can design an AI based on this approach, we have to define how humans think. We either have to study the thoughts or the physiology of the human brain, which is obviously harder. In computer science, one result of the cognitive modeling approach is the General Problem Solver (GPS) (Newell and Simon, 1961)[6].

2.1.3. Thinking rationally

This approach is based on the laws of thought based on **logic**. Logic is based on **syllogisms** by Greek philosopher Aristotle. In the 19th century, logistics defined a precise notation for statements about the world around us and relation between them.

However, there are two problems with this approach. First, representing all problems in the world by this notation could be incredibly complex, and second, information can often be incomplete and thus hard to solve with 100% correctness.

2.1.4. Acting rationally

This approach is based on the concept of **agents**. An agent is an entity that acts, but computer agents also have other attributes. Agents that decide how to act based on previous experiences stored in a memory and the program are called **rational agents**.

Rational agents, however, do not have to be completely correct as the "laws of thought"approach requires. The agent might take a suboptimal action, either because there is no other viable option for the agent (for instance, reflex) or because there might be another gain in the future (searching phase).

This approach has two advantages. It can be applied to more general problems than the "laws of thought"approach, and compared to the approached based on the human mind, it is broader and can be applied to entities that are not human in nature.

2.2. Multi-agent Systems

The goal of this thesis is to write a multi-agent systems simulator in Common Lisp. Therefore, as defined in the previous section, such a simulator would fall under the *acting rationally* approach. For that, we need to precisely define what a multi-agent system is.

A **multi-agent system** is a system containing an **environment** and one or more **agents**. These agents perceive the environment via **sensors** and modify it via **actuators**. The function that defines which actuators the agent will use to modify the environment is called the **agent function**.

2.3. Agents

An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through its **actuators** (Russel and Norvig, 32)[1]. We define a **percept** ϕ as the current value of one of its sensors at the current time. A **percepts sequence** Φ is a history of all percepts that the agent has perceived until now. The behavior of an agent is defined by its **agent function** f, which takes in a percept sequence and percept and returns a set of actuators A_a and a modified percept sequence Φ_m:

$$f(\Phi, \phi) \longrightarrow \{\Phi_m, A_a\} \mid \phi \in \Phi_m$$

An **actuator** a is a function that takes environment E and returns modified environment E_m:

$$a(E) \longrightarrow E_m$$

Thus we can define an agent α as a set containing a set of sensors Σ, a set of actuators Γ and an agent function f:

$$\alpha = \{\Sigma, \Gamma, f\}$$

An **agent program** is a concrete implementation of a mathematical **agent function**.

2.3.1. Rational agent

For an agent to be a rational agent, it must behave correctly, i.e. it does the right thing at every moment. However, for that we need to define a way to measure success. A **performance measure** determines whether an agent has performed successfully or not. During the lifetime of an agent, it will generate changes to the environment via its actuators. Based on those changes, the performance measure determines how successful the agent was. A performance measure must not be part of the agent itself, but must instead be created by the author of said agent, because it needs to be objective.

Rationality Rationality can be defined on these four criteria:

- Existence of a performance measure
- Prior knowledge of the environment it is in
- Actions that an agent can perform
- Percept sequence of percepts to date

Then we can define a rational agent as: *For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percepts sequence and whatever built-in knowledge the agent has.*(Russel and Norvig, 36)[1]

Other attributes of a rational agent An agent must not be **omniscient**, i.e. it must not know the actual outcome of its actions. It might predict one, but it must not be 100% accurate. Rationality maximizes the expected performance, while perfection maximizes its actual performance. There is no need to create perfect, omniscient agents, because they can not exist in real world environments, only in very limited, controlled ones.

A rational agent might also **learn** during its existence. Learning is basically modifying its agent function based on the percepts sequence. Prior to that, there might also be a phase of **exploration**, where an agent does not perform its normal duty, but instead gathers as many percepts as it can because it wants to modify its agent function with learning. If an agent is capable of learning, we can express his agent function as:

$$f(\Phi, \phi) \longrightarrow \{\Phi_m, A_a, f_m\} \mid \phi \in \Phi_m$$

If an agent relies on internal knowledge that its author gave to it and not by the percept sequence it has gathered so far, we can say that agent lacks **autonomy**.

2.4. Agent types

We can divide agents into types based on their agent function:

Table agent A table agents' agent function contains a table of percept sequences and actuators. Every history of percepts is mapped exactly to a list of actuators. While such an agent is perfect, it is unrealistic in any environment, except minimalistic.

Reflex agent A reflex agent is an agent whose agent function decides on actuators only by the list of current percepts, ignoring the percept sequence entirely.

Model-based agent A model-based agents' agent function uses knowledge of the environment gathered by percept sequence along with its current list of percepts to determine the actuators.

Goal-oriented agent A goal-oriented agent is the same as a model-based agent, but also operates on the notion of a goal it is trying to achieve. It might even decrease its performance measure to fulfill it.

Utility agent Utility agents operate with the notion of happiness. Every action is determined by how happy it makes the agent, and such an agent tries to maximize the happiness it receives.

2.5. Environment

An **environment** is basically a set of constraints, rules and objects with which the agent is expect to interact. An environment has multiple properties:

Observability The environment is **fully observable** if an agent can, at any time, get a complete snapshot of the environment via its sensors. If some information is missing, the environment is only **partially observable**.

Determinism If the next state of the environment is fully defined by its current state and the set of actuators the agent produces via its agent function, it is **deterministic**. Otherwise, it is **stochastic**. However, if the next state of the environment is dependant on its previous state and all the actions of all the agents, it is called **strategic**.

Time continuum If the environment is **episodic**, actions can be divided into stand alone episodes where the agent reads the environment then performs any actions. However, this episode does not have any impact on any next or previous episodes. If an environment is not episodic, it is **sequential**.

State If the environment can change while agent is inside the agent function, such environment is then called **dynamic**. Otherwise it is called **static**.

State continuum If time is stopped when an agent queries the environment for its percepts, we can call the environment **discreet**. Otherwise it is **continuous**.

Agent count Lastly, if there is only one agent, we call the environment a **single agent** environment. Otherwise it is called a **multi-agent** environment. If agents in the multi-agent environment cooperate, it is called **cooperative**. Otherwise, it is called **competitive**.

3. Common Lisp language

The goal of this thesis is to create a multi-agent simulator in Common Lisp. It must conform not only to the basic theory presented in previous chapters but must also be flexible and easy to use.

3.1. Why Common Lisp?

Common Lisp has always been regarded as a language that artificial intelligence researchers should use. That misconception, however, is false. Common Lisp is a language that can be used by any programmer to create any application. Nevertheless, some parts of the language do have features which are helpful in artificial intelligence research.

3.1.1. What is Common Lisp

Common Lisp is a dialect of the Lisp language family covered in the ANSI Standard[1]. It is a strong[2] dynamically[3][4] typed language with both lexical[5] and dynamic[6][7] scoping.

Generally, Common Lisp is an interpreted language[8] that might be compiled into byte code or machine code before execution (JIT[9]). Common Lisp is cross platform; however, specific implementations might have platform-dependent fast loading modules containing compiled Lisp code.

Common Lisp is a multiparadigim language, which means it supports multiple ways of writing programs within it. You can write it functionally; use a powerful CLOS[10] to write object-oriented software; or use a powerful set of macro tools to perform source-to-source transformations, source-to-same-source check up and even modify the reader itself via reader macros.

3.1.2. Main features of Common Lisp

A lot of features of Common Lisp have been "reinvented" in a lot of the modern languages. However, Common Lisp still has many features which are never presented together in other programming languages. In this section of the thesis, the important features will be presented along with the flaws or common pitfalls that programmers should pay attention to.

[1] ANSI INCITS 226-1944 (R2004), formerly X3.266-1994 (R1999)

[2] Typing errors are prevented during runtime.

[3] Dynamically-typed languages check the types of variables during runtime by storing a tag containing information about the variable's type along with its value.

[4] However, Common Lisp can also use a static type system when desired by the programmer, where the compiler could exclude the tags for optimization.

[5] Lexical scoping means that variables are bound by the source code, preserving the closures as they are defined.

[6] Dynamic scope means that the content of a variable is defined by the runtime of the program.

[7] Common Lisp can use *special* variables which are dynamic.

[8] Interpreted as in loaded from `.lisp` source files instead of being compiled into machine code

[9] Just-in-time compilation, see more at http://en.wikipedia.org/wiki/Just-in-time_compilation

[10] Common Lisp Object System

Macros In my opinion, the most important feature of Common Lisp is its powerful macro system. A **macro** is an ordinary Common Lisp function in which arguments and return value are Common Lisp source code. This feature is only possible because of the so called "parenthetic" way of writing Common Lisp source code. Due to the fact that the source code of Common Lisp expression is all composed of regular Lisp objects, such as lists, symbols and such, macro functions can take that source code in unevaluated form and perform translations on it. The macro function then returns an expression itself which, if the code is evaluate, is immediately evaluated.

Example 1 Macro example

```
(defmacro if* ((if-variable-capture if-argument) true &optional false)
  `(let ((,if-variable-capture ,if-argument))
     (if ,if-variable-capture
         ,true
       ,false)))

> (if* (c (+ 4 5))
>      (print c))
9
9
```

There are two common problems with Common Lisp macros: multiple evaluation and identifier capture.

Multiple evaluation errors occur when we have a common part of the source copied over multiple places instead of evaluating once to a symbol and then using that value in the multiple places.

Example 2 Multiple evaluation error

```
(defmacro log-around (message &body code)
  `(progn
      (format t ,message)
      ,@code
      (format t ,message)))

(defun rnd-message ()
  (format nil "Random message: ~R~%" (random 1000)))

> (log-around (rnd-message)
      (print "Inside"))
Random message: seven hundred and twenty-eight

"Inside" Random message: four hundred and ninety-four
NIL
```

We could fix it by making a symbol `sym` and evaluating the value into it:

Example 3 Multiple evaluation (almost) correct code

```
(defmacro log-around-correction-1 (message &body code)
  `(let ((sym ,message))
     (format t sym)
```

```
      ,@code
      (format t sym)))

> (log-around-correction-1 (rnd-message)
      (print "Inside"))
Random message: two hundred and twenty-seven

"Inside" Random message: two hundred and twenty-seven
NIL
```

However, by trying to solve the previous error by binding the value into the symbol `sym`, we have created another problem. We introduced a new symbol into the old source code. Had the original source code contained symbol `sym` anywhere, our new `sym` would interfere with the original one. You might argue that you could use symbol `some-very-unlikely-to-be-really-used-symbol` instead of the generic `sym`; however, that is not the correct solution to the problem. Common Lisp gives us the tool we need for this, the function `gensym`, which creates a symbol that is guaranteed to never be used by the original source code or any other call to gensym.

Example 4 Multiple evaluation - correct code

```
(defmacro log-around-correct (message &body code)
  (let ((sym (gensym)))
    `(let ((,sym ,message))
       (format t ,sym)
       ,@code
       (format t ,sym))))

> (log-around-correct (rnd-message)
      (print "Inside"))
Random message: three hundred and thirty-one

"Inside" Random message: three hundred and thirty-one
NIL
```

Macros can also be used to inspect the source code, in which case they only traverse the expression (which is always also an AST[11]) and return it back afterwards.

CLOS Th CLOS - Common Lisp Object System - is a facility for object-oriented programming (OOP) in Common Lisp. CLOS differs from many other OOP facilities with its flexibility and robustness. The basic parts of CLOS are classes, instances of classes, generic functions and methods of those generic functions. In CLOS, a class is only an object with a list of slots and additional metadata. Classes can have multiple bases, and slots within classes can be allocated inside instances (with `:allocation :instance`, which is also default) or "statically"inside the class type (with `:allocation :class`).

CLOS is multi-dispatching system. Instead of grouping functions into methods bound by specific classes as in most OOP languages, CLOS uses generic functions which group concrete method implementations. A dispatch is then based on the types of arguments during runtime.

[11] Abstract Syntax Tree

Each concrete method lists the types of its arguments (or no type if it accepts any type for that argument), and the generic function itself dictates which method is actually called. Other than type, CLOS also allows `(eq _constant_)` as an argument, in which case the dispatch will fire that method if the provided argument is a constant. This is mostly used with `NIL`. CLOS dispatching is not only based on primary methods (standard methods), but also auxiliary methods. If you prefix a method with `:before`, `:after`, or `:around` you can register methods that are called before, after or around the primary method. This feature has been re-invented by aspect-oriented programming, especially in Java[12].

Restarts Most modern languages[13] have facilities included for handling exceptional states. Common Lisp has a similar facility included called Conditions. What makes Common Lisp's unique, however, is that it also provides the programmer a way to set up so called "restarts."

In normal exception handling, you have code logic broken into two parts: *signaling* an exception and *handling* the exception, with both parts handled by completely different code, usually across libraries. The Condition system, however, splits the code logic into three parts: *signaling* a condition, *handling* the condition and *restarting* from a condition.

In exception handling, if an exceptional state has occurred inside the code, all of the stack information between signaling and handling the exception is lost. However, with conditions, you can handle the condition (fix a problem, reestablish connection, etc) and then continue from the restart point, which could be back at the place where the exception was signaled, without losing the stack during the handling of that condition.

3.1.3. Why use the LispWorks Common Lisp implementation

LispWorks is multi-platform integrated development environment (IDE) for Common Lisp. I have decided to make `cl-ass` in LispWorks because:

- It is multi-platform, working on Linux, Windows, Mac Os X and many more
- It is fast, robust, and reliable
- It follows ANSI standard, including good MOP[14] support
- Personal use is only restricted by session time (you can only have it open for 5 hours, but you can reopen it to restart the timer)
- Multi-platform GUI toolkit CAPI
- LispWorks also provides FFI[15] for opening C/C++ based libraries
- `cl-ass` is supposed to be used by programmers studying artificial intelligence - having an IDE already included for free

[12] For instance, AspectJ.

[13] Including Java, C++, Python, etc.

[14] Meta Object Protocol - a way of implementing CLOS that was unfortunately not included in the ANSI standard

[15] The Foreign Function Interface

The main disadvantage of LispWorks is that it is a commercial product. To fully use its features, you have to pay for a license. However, `cl-ass` does not use any features not included in the free personal version of LispWorks (as it was programmed in one).

4. Common Lisp Multi-agent Systems Simulator - cl-ass

`cl-ass` is multi-platform, multi-agent systems simulator written in Common Lisp. Its main purpose is to present artificial intelligence researchers a way to easily create multi-agent simulations with an easy-to-use API that is based on the theory behind rational agents presented in chapter 2[2., page 9]. It supports saving a whole simulation into a single file, making it easily portable between computers. With a powerful IDE behind LispWorks, artificial intelligence researchers can use the tools included, such as a debugger, tracer, profiler or any library included with/available for Common Lisp.

4.1. Features of the cl-ass framework

`cl-ass` is based on the theory behind artificial intelligence. Therefore, the user can:

- Define an **environment**. `cl-ass` only supports **static state**[2.5., page 12], **continuous**[2.5., page 12] environments. By default, the environment is also **deterministic/strategic**[2.5., page 12]; however, the user can change that behavior by modifying the environment independent of agents.
- Define **agents**, their **percepts**, **actuators** and their **agent function**.
- Iterate over the simulation or reset it back to its starting state.
- Create a graphical window and display the contents of the simulation on the screen at every step.
- Export the simulation along with any of its dependencies into a separate lisp file[16].
- Use the provided abstraction to define a simulation and its components in a clean way, minimizing boilerplate coding.

`cl-ass` has been thoroughly tested on the Microsoft Windows 7 operating system; however, it should fully support every operating system that LispWorks supports with a CAPI interface (if you require the graphical portion of the framework), which includes at least Windows, Mac Os X and Unix/Linux.

4.2. Implementation

`cl-ass` consists of two `.lisp` files – `cl-ass.lisp` containing the main code of the framework and `capi-loader.lisp` containing the GUI add-on based on CAPI – along with 3 examples. To load the examples, you need to set the current working directory for LispWorks to the examples directory[17].

[16] Simulation code should not use any functions nor should it compile its methods. If you want to include functions, you can add the file as a dependency

[17] `(hcl:change-directory` *path*`)`

`cl-ass`'s main feature is its conformity with the theoretical basis of multi-agent systems. Agents are instances of the `agent` subclass, and both percepts and actuators are present within the agent as methods of the class. Additionally, percepts are present as slots of the agent class, thus, they can be easily queried during the agent function. Everything is correctly encapsulated, thus, you cannot (without using global variables) query the environment. Instead, agents have to rely on their internal memory and the values of percepts. Agent functions only need to return a list of symbols with the same name as actuators to perform those actions. Again, the environment is not available for modification outside the defined actuators.

4.3. Features of Common Lisp used

Common Lisp has not only been chosen because of how the end user will use the `cl-ass` framework, but also because implementing it without Common Lisp would have been much more difficult. Common Lisp contains various language constructs unique to it, as explained in chapter 3[3.1.2., page 14]. In this section, the thesis will present the parts of the Common Lisp language used in `cl-ass`.

Macros In `cl-ass`, macros are used for two purposes: general black-boxing of syntax, such as the macro `random-dolist`, or as a way of providing the end user with a way to define new constructs, such as `defsimulation`[5.3.3., page 31], `defagent`[5.3.2., page 28] and so on.

There is not a single programming language with as much macro flexibility as Common Lisp or other Lisp-derived languages. In fact, macros are so powerful that in the Scheme language, they tried to come up with a limited way to generate only syntax macros called hygienic macros. Some languages have preprocessor macros[18]; however, those are very limited[19], operating on their own syntax and performing only basic source code control.

CLOS The whole `cl-ass` framework is object-oriented with agents, environments and simulations being represented by classes. However, in the `capi-loader` portion of the framework, advanced method combinations along with auxiliary methods have been used to provide the simulation's counting features and forcing a refresh after running an iteration of it. These make the code much more readable and are called regardless of what the main method does, giving freedom to the user to define their main method however they wish while keeping the same functionality.

A similar thing has been added to the Java language as the AspectJ library. The library provides aspects, which are generally much simpler auxiliary methods. It has a few quirks on its own, because it needs to replace original classes with proxies, which makes debugging really painful, and is also not as powerful as auxiliary methods in CLOS.

MOP The whole section of the framework from `display-defclass` to `generate-specialized-arglist` was taken from the book "The Art of Metaobject Protocol"by Kiczales, Riveres and Bobrow[5] which contained functions to query classes, their generic functions and methods. Because LispWorks conforms to the MOP standard,

[18] Like C, C++

[19] The C/C++ preprocessor is not even Turing complete.

it was possible to create a generalized export function that works with any user-defined classes, as long as they are derived from `storable-class`[5.3.4., page 32].

Meta-classes are included in some scripting languages, such as Python, which is one of the reasons Python is so flexible[20].

Dynamic variables The `*in-export*` symbol is used as a special variable with dynamic binding. It is used in the exporting function to prevent opening new interfaces for CAPI based simulations. Similarly in `export-simulation`, the `*standard-output*` symbol is rebound to a file, saving us the need to pass along the reference to a stream for every function called. Instead, we print to standard output.

This is possible in other languages by using a global variable with a hash map, using the thread ID as the key and its value as the value, querying it with the current thread ID and getting/setting the variable. However, it is much more cumbersome than simply setting and getting the value out of the symbol. It also needs to be made thread-safe by the programmer while dynamic variables in Common Lisp are already thread-safe. Additionally, it presents the problem of having global variables in the program, while dynamic variables in Common Lisp do not need to be declared globally – instead, they can be declared as special inside the function.

Feature symbols, programmable reader Common Lisp is also unique in the fact that you can change how the reader reads the source code. In `cl-ass`, the `:cl-ass` keyword is used to store the fact that cl-ass has been loaded, preventing it from being loaded again by `capi-loader`. Additionally, the `:quicklist` keyword is queried at the beginning of `cl-ass` to check for quicklisp. If it is present, the cl-store library is loaded, and the ability to export running simulations is enabled.

Similar functionality is available in C/C++ via macros. The C/C++ pre-processor uses defines to control which part of the source code is available. Its problem is the same as with macros in Lisp: it is a separate feature and not controlled by the C/C++ language. Moreover, the programmable reader of Common Lisp is infinitely more powerful, with possibilities of reading in-line Lisp within HTML[21].

[20]Python has been inspired by the Lisp language family in more aspects, such as lambdas, specific loop constructs similar to `dolist`, etc.

[21]I have made a prototype Lisp web server with in-line Lisp called QUAD: https://github.com/Enerccio/Quad

5. User Manual

5.1. Installation and running the examples

5.1.1. Installation

`cl-ass` does not require any installation. To use the library, you just need to load the included files `cl-ass.lisp` and (optionally) `capi-loader.lisp` (in any order; however, if you load capi first, you need to have the Lisp path set up to the same directory where cl-ass is[22]).

`cl-ass` can, optionally, use `cl:store` for serializing the framework. If you want to use `cl:store`, you need to have `quicklisp`[23] ready and loaded before loading `cl-ass`. You may opt out of this dependency by not having quicklisp; however, you will not be able to store the current state of the simulation during the export.

5.1.2. Loading examples

Every example requires the Lisp path to be set to the directory where the example file is located, and that directory must be inside the directory where cl-ass (and optionally, capi-loader) reside to correctly auto-load the dependencies.

Three examples are provided with the `cl-ass` framework:

- `example-cli.lisp` - Simple reflexive agent, simulating an automated vacuum cleaner, with no GUI
- `example-ma-cli.lisp` - Swarm intelligence, with automatic closest path searching ant-agents
- `example-gui.lisp` - Same simulation as in `example-cli.lisp`, with a GUI

5.2. How to use

In this section, a concrete agent simulation based on Simple Reflex Agents[1] is presented as an example of how to use the cl-ass framework.

5.2.1. Environment

Every simulation must contain a defined environment. In our example, the environment is a simple, flat, 2-dimensional land with randomly-generated clean or dirty tiles. We define the environment via the `defenvironment` macro [5.3.1., page 27]. Our dirty flatland only contains the world (2D array) and the current position of our agent as its slots.

[22] `(hcl:change-directory` *path*`)`
[23] http://www.quicklisp.org/beta/

Example 5 Defining new environment

```
(defenvironment dirty-flatland
                ((current-agent-position :initform 0 :accessor agent-position)
                 (flatland :initform nil :accessor flatland)))
```

We need to provide a method for this environment that will take care of initializing every new instance of it.

Example 6 Method for initializing new environment

```
(defmethod initialize-environment ((e dirty-flatland) s)
  (let* ((len (+ 5 (random 4)))
         (pos (random len)))
    (let ((vec (make-array len)))
      (dotimes (i len)
        (setf (aref vec i)
              (if (= (random 2) 0)
                  :dirty
                :clean)))
      (setf (agent-position e) pos)
      (setf (flatland e) vec))))
```

We should also provide methods for our agent to later use to modify the environment. `agent-tile` is used to return where our agent is standing in our 2D world. `clean-agent-tile` will clean the current tile where our agent is standing, and methods `left` and `right` will move the agent left or right (or not at all, if it is at the edge of our 2D world).

Example 7 Methods for modifying environment

```
(defmethod agent-tile ((e dirty-flatland))
  (aref (flatland e) (agent-position e)))

(defmethod clean-agent-tile ((e dirty-flatland))
  (setf (aref (flatland e) (agent-position e))
        :clean))

(defmethod left ((e dirty-flatland))
  (setf (agent-position e)
        (max 0
             (1- (agent-position e)))))

(defmethod right ((e dirty-flatland))
  (setf (agent-position e)
        (min (1- (length (flatland e)))
             (1+ (agent-position e)))))
```

5.2.2. Agent

Next, we need to define our agent classes. In our example, we have a single agent simulation, so we only need to define one agent class, namely **vacuum-cleaner**. We use the

macro `defagent`[5.3.2., page 28] to define our agent. Our agent will only have one defined slot, `cleaned-tiles`, where we will count the number of tiles that agent has cleaned so far.

Example 8 Defining our agent

```
(defagent vacuum-cleaner ((cleaned-tiles :initform 0 :accessor cleaned-tiles))
  )
```

Our agent needs to be able to query the environment. For that, we need to define a set of percepts. But since our agent is very simple, we only need one percept, namely `is-dirty?`, which queries whether the current tile is dirty. We can define our percept via the `defpercept` macro [5.3.2., page 29].

Example 9 is-dirty? percept definition

```
(defpercept is-dirty? ((agent vacuum-cleaner) e)
  (eq (agent-tile e) :dirty))
```

In order for our agent to modify the environment, it needs a set of actuators. We define four actuators via the `defactuator` macro[5.3.2., page 29]. `suck`, which cleans the current tile, `go-left` and `go-right` which move the agent in the environment, and `contemplate`, which makes our agent do nothing this iteration.

Example 10 Defining actuators

```
(defactuator suck ((agent vacuum-cleaner) e)
  (clean-agent-tile e))

(defactuator go-left ((agent vacuum-cleaner) e)
  (left e))

(defactuator go-right ((agent vacuum-cleaner) e)
  (right e))

(defactuator contemplate ((agent vacuum-cleaner) e)
  )
```

The final thing we need to define for our agent is its `agent-function` method[5.3.2., page 30]. This function will be the main logic hub of the agent. We need to return lists of actuators out of every branch of our agent function.

Example 11 Simple Vacuum Cleaner agent function

```
(defmethod agent-function ((agent vacuum-cleaner))
  (with-slots (is-dirty?) agent
    (cond (is-dirty? (incf (cleaned-tiles agent))
                     (list 'suck))
          (t (let ((a (random 3)))
               (cond ((= a 0) (list 'contemplate))
                     ((= a 1) (list 'go-left))
                     (t (list 'go-right))))))))
```

5.2.3. Simulation

At the end, we define our simulation via the `defsimulation` macro[5.3.3., page 31]. We provide a name, an empty list of slots, a name for our environment class, and a list of agent classes - in our case, only a single `vacuum-cleaner` agent.

Example 12 Defining simulation

```
(defsimulation vacuum-cleaner-simulation ()
                        dirty-flatland (vacuum-cleaner))
```

We may now use our simulation to simulate our agent in our environment. For that, we need to make an instance of our simulation, initialize it via `initialize-simulation`[5.3.3., page 31] and then iterate over simulation steps via the `iterate` method[5.3.3., page 32].

Example 13 Running the simulation

```
> (setf *simulation* (make-instance 'vacuum-cleaner-simulation))
#<VACUUM-CLEANER-SIMULATION 20093A1B>

> (initialize-simulation *simulation*)
#<VACUUM-CLEANER-SIMULATION 20093A1B>

> (iterate *simulation*)
NIL

> (iterate *simulation* 1000)
NIL
```

We can also query information about our simulation.

Example 14 Querying the simulation

```
> (agents *simulation*)
(#<VACUUM-CLEANER 2198BD2B>)

> (cleaned-tiles (first (agents *simulation*)))
4

> (agent-position (environment *simulation*))
1

> (flatland (environment *simulation*))
#(:CLEAN :DIRTY :DIRTY :CLEAN :DIRTY :CLEAN :CLEAN)
```

5.2.4. Complete example

Here is the complete source code to the vacuum cleaning agent presented in previous sections.

Example 15 Random vacuum cleaning agent simulation

```
(defenvironment dirty-flatland
```

```
                ((current-agent-position :initform 0 :accessor agent-position)
                 (flatland :initform nil :accessor flatland)))

(defmethod initialize-environment ((e dirty-flatland) s)
  (let* ((len (random 1000))
         (pos (random len)))
    (let ((vec (make-array len)))
      (dotimes (i len)
        (setf (aref vec i)
              (if (= (random 2) 0)
                  :dirty
                :clean)))
      (setf (agent-position e) pos)
      (setf (flatland e) vec))))

(defmethod agent-tile ((e dirty-flatland))
  (aref (flatland e) (agent-position e)))

(defmethod clean-agent-tile ((e dirty-flatland))
  (setf (aref (flatland e) (agent-position e))
        :clean))

(defmethod left ((e dirty-flatland))
  (setf (agent-position e)
        (max 0
             (1- (agent-position e)))))

(defmethod right ((e dirty-flatland))
  (setf (agent-position e)
        (min (1- (length (flatland e)))
             (1+ (agent-position e)))))

(defagent vacuum-cleaner ((cleaned-tiles :initform 0 :accessor cleaned-tiles))
   )

(defpercept is-dirty? ((agent vacuum-cleaner) e)
  (eq (agent-tile e) :dirty))

(defactuator suck ((agent vacuum-cleaner) e)
  (clean-agent-tile e))

(defactuator go-left ((agent vacuum-cleaner) e)
  (left e))

(defactuator go-right ((agent vacuum-cleaner) e)
  (right e))

(defactuator contemplate ((agent vacuum-cleaner) e)
  )

(defmethod agent-function ((agent vacuum-cleaner))
  (with-slots (is-dirty?) agent
    (cond (is-dirty? (incf (cleaned-tiles agent))
                     (list 'suck))
```

```
          (t (let ((a (random 3)))
               (cond ((= a 0) (list 'contemplate))
                     ((= a 1) (list 'go-left))
                     (t (list 'go-right))))))))

(defsimulation vacuum-cleaner-simulation ()
                          dirty-flatland (vacuum-cleaner))
```

5.3. Full API Documentation

5.3.1. Environment

Class **ENVIRONMENT-CLASS**

Class Precedence List:

storable-class[5.3.4., page 32] **standard-class t**

Class **ENVIRONMENT**

Class Precedence List:

environment-class storable-class[5.3.4., page 32] **standard-class t**

Macro **DEFENVIRONMENT**

Syntax:

defenvironment *name ({slot-specifier}*)*

⇒ *new-class*

```
slot-specifier::= slot-name | (slot-name [[slot-option]])
slot-name::= symbol
slot-option::= {:reader reader-function-name}* |
               {:writer writer-function-name}* |
               {:accessor reader-function-name}* |
               {:initform form} |
               {:type type-specifier} |
               {:documentation string}
function-name::= {symbol | (setf symbol)}
```

Arguments and Values:

name a non-nil symbol

slot-name a symbol that is syntactically valid for use as a variable name

reader-function-name a non-nil symbol

writer-function-name a generic function name

form form

type-specifier a type specifier

Macro **defenvironment** defines a new environment and returns it as a new class. This new class will have `environment`[5.3.1., page 27] as its superclass.

```
(defenvironment house
   ((rooms :initform (generate-rooms) :accessor rooms)))
```

Generic Function **initialize-environment**

Syntax:

initialize-environment *environment simulation*

⇒ *undefined*

`Initialize-environment` is a method that is called every time a simulation is initialized and should prepare the underlying environment for its execution. Every environment must have at least one `initialize-environment` method defined.

```
(defmethod initialize-environment ((env house) simulation)
     ... )
```

5.3.2. Agent

Class **AGENT-CLASS**

Class Precedence List:

storable-class[5.3.4., page 32] **standard-class t**

Class **AGENT**

Class Precedence List:

agent-class storable-class[5.3.4., page 32] **standard-class t**

Macro **DEFAGENT**

Syntax:

defagent *name ({slot-specifier}*)*

⇒ *new-class*

```
slot-specifier::= slot-name | (slot-name [[slot-option]])
slot-name::= symbol
slot-option::= {:reader reader-function-name}* |
               {:writer writer-function-name}* |
               {:accessor reader-function-name}* |
               {:initform form} |
               {:type type-specifier} |
```

```
                {:documentation string}
function-name::= {symbol | (setf symbol)}
```

Arguments and Values:
name a non-nil symbol
slot-name a symbol that is syntactically valid for use as a variable name
reader-function-name a non-nil symbol
writer-function-name a generic function name
form form
type-specifier a type specifier

Macro **defagent** defines a new agent and returns it as a new class. This new class will have `agent`[5.3.2., page 28] as its superclass.

```
(defagent mouse
  ((stomach :initform :empty :accessor stomach)
   (bravery :initform 0 :accessor bravery)
   (alive :initform t :accessor alive-p)))
```

Macro **DEFPERCEPT**
Syntax:
defpercept *name ((variable agent-type) environment) form**
⇒ *new-method*

Arguments and Values: *name* a symbol that is syntactically valid for use as a method name
variable a symbol that is syntactically valid for use as a variable
agent-type type
environment a symbol that is syntactically valid for use as a variable
form form

Macro **defpercept** defines a new percept for an agent. A percept is a method bound for the type specified by agent-type. The body of the method defined by **defpercept** must return some value which should be obtained from the environment. Additionally, every percept defined for the agent adds a new slot to the agent class with the same name as the name argument. This can be used later, in `agent-function` to query the percept value.

```
(defpercept near-cheese? ((agent mouse) env)
   (has-items (get-room-of env agent) :cheese))
```

Macro **DEFACTUATOR**
Syntax:
defactuator *name ((variable agent-type) environment) form**
⇒ *new-method*

Arguments and Values: *name* a symbol that is syntactically valid for use as a method name
variable a symbol that is syntactically valid for use as a variable

agent-type type
environment a symbol that is syntactically valid for use as a variable
form form

Macro **defactuator** defines a new actuator for an agent. The actuator is a method bound for the type specified by agent-type. The body of the method defined by **defactuator** can modify the environment and represents actions that the agent wants to take.

```
(defpercept eat-cheese ((agent mouse) env)
   (remove-item (get-room-of env agent) :cheese))
```

Generic Function **AGENT-FUNCTION**

Syntax:
agent-function *agent*
⇒ *list of actuators*

Arguments and Values:
list of actuators nil or a list containing symbols which are defined as actuators for this agent
`agent-function` is the main method of an agent. In this method, the agent should query slots with the same names as the percepts defined for that agent and decide which actions to take. Actions are then returned as a list of actuators.

```
(defmethod agent-function ((agent mouse))
  (with-slots (near-cheese? north? south? east? west? cat-in-room?) agent
     (cond (((and near-cheese? (not cat-in-room?))
                   '(eat-cheese))
                  (cat-in-room?
                     (...
        ...)
```

5.3.3. Simulation

Class **SIMULATION-CLASS**

Class Precedence List:

storable-class[5.3.4., page 32] **standard-class t**

Class **SIMULATION**

Class Precedence List:

simulation-class storable-class[5.3.4., page 32] **standard-class t**
Slots:

environment, **accessor**: *environment* - an instance of the current environment
agents, **accessor**: *agents* - the list of all currently instantiated agents

Macro **DEFSIMULATION**
Syntax:
defsimulation *name ({slot-specifier}*) environment-class ({agent}*)*
⇒ *new-class*

```
slot-specifier::= slot-name | (slot-name [[slot-option]])
slot-name::= symbol
slot-option::= {:reader reader-function-name}* |
               {:writer writer-function-name}* |
               {:accessor reader-function-name}* |
               {:initform form} |
               {:type type-specifier} |
               {:documentation string}
function-name::= {symbol | (setf symbol)}
```

Arguments and Values:
name a non-nil symbol
slot-name a symbol that is syntactically valid for use as a variable name
reader-function-name a non-nil symbol
writer-function-name a generic function name
form form
type-specifier a type specifier
environment-class a type specifier
agent a type specifier
Macro **defsimulation** defines a new simulation and returns it as a new class. Every simulation defined by defsimulation will have `simulation`[5.3.3., page 30] as its superclass.

```
(defsimulation cat-and-mouse
  ((iteration-count :initform 0 :accessor it-count))
  house
  (mouse mouse mouse cat cat))
```

Generic Function **INITIALIZE-SIMULATION**
Syntax:
initialize-simulation *simulation*
⇒ *simulation*

Method signatures:
initialize-simulation *(s simulation)*
Arguments and Values:
simulation a simulation instance

Method initialize-simulation should be called when you want to reset the simulation to a pristine state. A new environment and agents are instantiated and generated.

```
> (initialize-simulation *simulation*)
#<CAT-AND-MOUSE 200E59F7>
```

Generic Function **ITERATE**

Syntax:
iterate *simulation* &optional *(n 1)*
⇒ *NIL*

Method signatures:
iterate *(s simulation)*
Arguments and Values:
simulation a simulation instance
n an integer

Iterates the simulation n times.

```
> (iterate *simulation*)
NIL
```

5.3.4. Export

Class **STORABLE-CLASS**

Class Precedence List:

standard-class t
Slots:

import-file, **accessor**: *import-file* - list of strings

Every class that you want to be able to export should have `storable-class` as one of its superclasses. In the import-file, you can store additional source dependencies which will be copied into the exported file.

Function **EXPORT-SIMULATION**

Syntax:
export-simulation *export-file* &key *simulation-instance (path ".")* ⇒ *NIL*

Arguments and Values:
export-file a string with the file name to which you want to export the simulation (will be overwritten)
simulation-instance a simulation instance. If provided, it will be stored in the

`deserialize-saved-state` function at the end of file
path a string. It is combined with import-file dependencies of all storable-classes

This functions takes all `storable-classes` defined in the current Lisp environment and defines which files these classes require by querying import-file. It then combines all these files into a single output file specified by the export-file argument. Then it exports all class definitions which have `storable-class` as their class precedence list along with all their methods, provided they are evaluated and not compiled. If the simulation-instance argument is provided, it is stored via cl:store into a specific function `deserialize-saved-state`.

Function **DESERIALIZE-SAVED-STATE**
Syntax:
export-simulation ⇒ *simulation-instance*

Arguments and Values:
simulation-instance a simulation instance

This functions returns an instance of a previously-exported simulation. This function is read-only and will always return a new instance with the same state that it had at the time of export.

```
> (deserialize-saved-state)
#<CAT-AND-MOUSE 200E80DE>
```

5.4. Graphics library

Included with cl-ass is a small library based on CAPI [24] included in LispWorks. The library is included in `capi-loader.lisp`.

5.4.1. How to use

Building off of the previous example, we only need to make minor changes to include graphics. Firstly, we need to define our simulation with `defgsimulation`[5.4.3., page 38] instead of `defsimulation`[5.3.3., page 31].

Example 16 Defining graphics simulation

```
(defgsimulation vacuum-cleaner-simulation ((graphics :accessor graphics :
    initform nil)) dirty-flatland (vacuum-cleaner) ())
```

We need to define the `clear-pane`[5.4.3., page 39] method. In it, during the first render call, we can load and convert images for the output pane.

[24] CAPI is a multiplatform GUI toolkit included with LispWorks. See more at http://www.lispworks.com/products/capi.html

Example 17 Defining clean-pane for our simulation

```
(defmethod clear-pane ((s vacuum-cleaner-simulation) pane)
 (unless (graphics s)
    (let ((r (render-panel s)))
      (setf (graphics s)
            (list (cons :clean   (gp:convert-external-image r (gp:
    read-external-image "images/empty.png")))
                  (cons :dirty   (gp:convert-external-image r (gp:
    read-external-image "images/dirty.png")))
                  (cons :c-agent (gp:convert-external-image r (gp:
    read-external-image "images/empty_agent.png")))
                  (cons :d-agent (gp:convert-external-image r (gp:
    read-external-image "images/dirty_agent.png"))))))))
  (gp:clear-graphics-port pane))
```

As a last thing, we need to define a way to render our simulation via the **render-pane**[5.4.3., page 39] method.

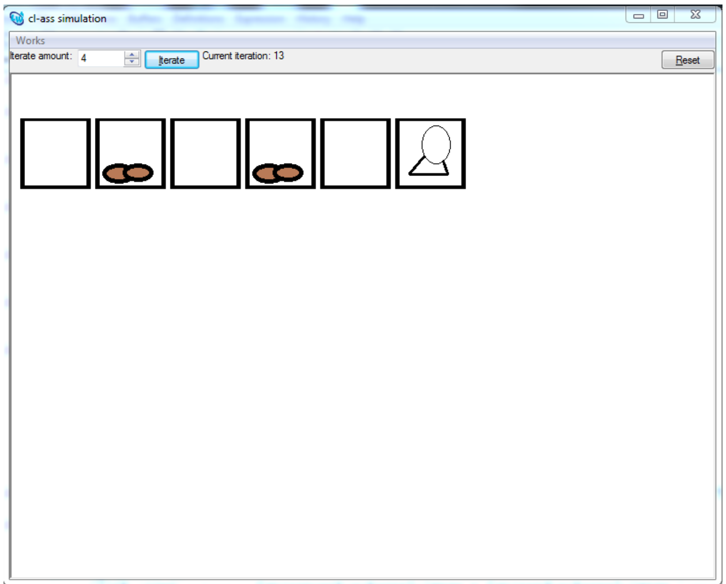
Example 18 Defining rendering

```
(defmethod render-pane ((s vacuum-cleaner-simulation) pane)
  (let ((e (environment s)))
    (let ((xpos 10))
      (dotimes (i (length (flatland e)))
        (let ((type (aref (flatland e) i)))
          (if (= i (agent-position e))
              (draw-agent s pane type xpos)
            (draw-empty s pane type xpos))
          (incf xpos 85))))))

(defun draw-agent (s pane type xpos)
  (draw-tile (assoc (if (eq type :dirty) :d-agent :c-agent) (graphics s)) pane
    xpos))

(defun draw-empty (s pane type xpos)
  (draw-tile (assoc type (graphics s)) pane xpos))

(defun draw-tile (img pane xpos)
  (gp:draw-image pane (cdr img) xpos 50))
```

We can then run the simulation by simply instantiating the defined simulation:

Example 19 Running the simulation

```
> (setf *simulation* (make-instance 'vacuum-cleaner-simulation))
#<VACUUM-CLEANER-SIMULATION 20093A1B>
```



Obrázek 1.: Vacuum Cleaner simulation in graphics.

5.4.2. Complete example

Here is the complete source code to the vacuum cleaning agent presented in the previous sections.

Example 20 Random vacuum cleaning agent simulation - graphics version

```
(defenvironment dirty-flatland
                ((current-agent-position :initform 0 :accessor agent-position)
                 (flatland :initform nil :accessor flatland)))

(defmethod agent-tile ((e dirty-flatland))
  (aref (flatland e) (agent-position e)))

(defmethod clean-agent-tile ((e dirty-flatland))
  (setf (aref (flatland e) (agent-position e))
        :clean))

(defmethod left ((e dirty-flatland))
  (setf (agent-position e)
        (max 0
             (1- (agent-position e)))))

(defmethod right ((e dirty-flatland))
  (setf (agent-position e)
```

```
        (min (1- (length (flatland e)))
             (1+ (agent-position e)))))

(defmethod initialize-environment ((e dirty-flatland) s)
  (let* ((len (+ 5 (random 4)))
         (pos (random len)))
    (let ((vec (make-array len)))
      (dotimes (i len)
        (setf (aref vec i)
              (if (= (random 2) 0)
                  :dirty
                :clean)))
      (setf (agent-position e) pos)
      (setf (flatland e) vec))))

(defagent vacuum-cleaner ((cleaned-tiles :initform 0 :accessor cleaned-tiles))
  )

(defpercept is-dirty? ((agent vacuum-cleaner) e)
  (eq (agent-tile e) :dirty))

(defactuator suck ((agent vacuum-cleaner) e)
  (clean-agent-tile e))

(defactuator go-left ((agent vacuum-cleaner) e)
  (left e))

(defactuator go-right ((agent vacuum-cleaner) e)
  (right e))

(defactuator contemplate ((agent vacuum-cleaner) e)
  )

(defmethod agent-function ((agent vacuum-cleaner))
  (with-slots (is-dirty?) agent
    (cond (is-dirty? (incf (cleaned-tiles agent))
                     (list 'suck))
          (t (let ((a (random 3)))
               (cond ((= a 0) (list 'contemplate))
                     ((= a 1) (list 'go-left))
                     (t (list 'go-right)))))))))

(defgsimulation vacuum-cleaner-simulation ((graphics :accessor graphics :initform nil)) dirty-flatland (vacuum-cleaner) ())

(defmethod clear-pane ((s vacuum-cleaner-simulation) pane)
 (unless (graphics s)
    (let ((r (render-panel s)))
      (setf (graphics s)
            (list (cons :clean   (gp:convert-external-image r (gp:read-external-image "images/empty.png")))
                  (cons :dirty   (gp:convert-external-image r (gp:read-external-image "images/dirty.png")))
                  (cons :c-agent (gp:convert-external-image r (gp:
```

```
    read-external-image "images/empty_agent.png")))
                   (cons :d-agent (gp:convert-external-image r (gp:read-external-image "images/dirty_agent.png")))))))
  (gp:clear-graphics-port pane))

(defmethod render-pane ((s vacuum-cleaner-simulation) pane)
  (let ((e (environment s)))
    (let ((xpos 10))
      (dotimes (i (length (flatland e)))
        (let ((type (aref (flatland e) i)))
          (if (= i (agent-position e))
              (draw-agent s pane type xpos)
            (draw-empty s pane type xpos))
          (incf xpos 85))))))

(defun draw-agent (s pane type xpos)
  (draw-tile (assoc (if (eq type :dirty) :d-agent :c-agent) (graphics s)) pane xpos))

(defun draw-empty (s pane type xpos)
  (draw-tile (assoc type (graphics s)) pane xpos))

(defun draw-tile (img pane xpos)
  (gp:draw-image pane (cdr img) xpos 50))
```

5.4.3. Graphics api

Class **SIMULATION-INTERFACE**

Class Precedence List:

capi:interface
Slots:
simulation, **accessor**: *simulation* - simulation instance

Graphics simulation uses this interface subclass as the main interface. It only has a single new slot, `simulation`, which contains the bound `graphics-simulation`[5.4.3., page 37] instance.

Class **GRAPHICS-SIMULATION**

Class Precedence List:

simulation simulation-class t
Slots:
init-interface-args, **accessor**: *init-interface-args* - initial arguments for interface initialization. These define how will the window look

iteration-count, **accessor**: *iteration-count* - counter for the number of iterations
interface, **accessor**: *interface* - slot containing `simulation-interface`[5.4.3., page 37] instance
render-panel, **accessor**: *render-panel* - output-pane instance which is used to render graphics

The subclass of the `simulation`[5.3.3., page 30] class which controls the graphical side of the simulation.

Macro **DEFGSIMULATION**

Syntax:
defgsimulation *name ({slot-specifier}*) environment-class ({agent}*) ({interface-args*})* ⇒ *new-class*

```
slot-specifier::= slot-name | (slot-name [[slot-option]])
slot-name::= symbol
slot-option::= {:reader reader-function-name}* |
               {:writer writer-function-name}* |
               {:accessor reader-function-name}* |
               {:initform form} |
               {:type type-specifier} |
               {:documentation string}
function-name::= {symbol | (setf symbol)}
```

Arguments and Values:
name a non-nil symbol
slot-name a symbol that is syntactically valid for use as a variable name
reader-function-name a non-nil symbol
writer-function-name a generic function name
form form
type-specifier a type specifier
environment-class a type specifier
agent a type specifier
interface-args plist of initializers. If not specified, `(:visible-min-width 800 :visible-min-height 600 :title "cl-ass simulation")` is used.

Macro **defgsimulation** defines a new graphics simulation and returns it as a new class. Every simulation defined by defgsimulation will have `graphics-simulation`[5.4.3., page 37] as its superclass. It will also initialize the simulation and show the window with it.

```
(defgsimulation cat-and-mouse
  ((iteration-count :initform 0 :accessor it-count))
  house
  (mouse mouse mouse cat cat)
  ())
```

Generic Function **MAKE-INTERFACE**

Syntax:

make-interface *graphics-simulation*

⇒ *output-pane*

Method signatures:

make-interface *(gs graphics-simulation)*

Arguments and Values:

graphics-simulation a graphics simulation instance

This method creates the content of the simulation interface.

Generic Function **CLEAR-PANE**

Syntax:

clear-pane *graphics-simulation output-pane*

⇒ *undefined*

Method signatures:

clear-pane *(gs graphics-simulation) output-pane*

Arguments and Values:

graphics-simulation a graphics simulation instance

output-pane rendering pane

This method is called before the rendering method to clear the content of the `output-pane`.

Generic Function **RENDER-PANE**

Syntax:

render-pane *graphics-simulation output-pane*

⇒ *undefined*

Method signatures:

render-pane *(gs graphics-simulation) output-pane*

Arguments and Values:

graphics-simulation a graphics simulation instance

output-pane rendering pane

This method is called when the simulation should render its content into the `output-pane`.

Conclusions

The aim of this thesis was to use Common Lisp to create a multi-agent systems simulator. Such a simulator should conform to the theoretical rules of multi-agent systems and should also be able to export the simulation into a separate file. The `cl-ass` framework conforms to the theoretical basis of multi-agent systems and has the ability to export source code into a separate file. Additionally with the powerful IDE powered by LispWorks, users can debug, trace, and profile their code. With `capi-loader`, users can also create simple graphical representations of their simulations, which allows them to visualize the state of an experiment. The complete API documentation has been provided along with three examples, two of which are dissected in the user manual, making the use of `cl-ass` as easy as possible. During its development, many features of Common Lisp have been used, and without them, it would have been very difficult, much more time consuming, and the framework itself would have been much bigger than it is now. Generally, working with Common Lisp has been an enjoyable experience, mostly due to its ability to change code on the fly and see the results, which I am sure will also be enjoyed by any user of the framework.

Závěr

Cieľom tejto bakalárskej práce bolo použiť jazyk Common Lisp ako nástroj pre vytvorenie frameworku pre multiagentné systémy. Simulátor by mal byť založený na teoretickom základ umelej inteligencie, špecificky multiagentných systémov. Rovnako by mal mať schopnosť exportu zdrojového kódu simulácie do nového súboru. `cl-ass` framework tento cieľ splĺňa v oboch bodoch. Naviac, použitím LispWorks ako vývojového prostredia, užívateľ frameworku dostane k rukám mocný nástroj v ktorom môže svoju simuláciu ladiť, profilovať a tak podobne. Okrem toho, v `capi-loader` je pripravený addon cez ktorý môže užívateľ jednoducho vizualizovat svoju simuláciu graficky. Súčasťou frameworku je i kompletná špecifikácia programovateľského rozhrania (API) ako aj tri ukážky použitia v praxi, z čoho dve su kompletne popísane v manuály. Počas vývoju frameworku bolo použitých veľa súčastí jazyka Common Lisp, bez ktorých, ak by bol framework vyvýjaný v inom jazyku, by to bolo oveľa pracnejšie, menej elegantné a i zdrojový kód by bol večší. Práca s jazykom Common Lisp bola veľmi príjemná, hlavne kôli možnosti za jazdy meniť kód a debugovať, čo som si istý, že využijú i budúci používatelia `cl-ass` frameworku.

Reference

[1] Russel S., Norvig P.: Artificial Intelligence: A Modern Approach, Pearson Education, New Jersey, 2003, ISBN 0-13-080302-2.

[2] Wooldridge M.: An Introduction to MultiAgent Systems, John Wiley & Sons Ltd, Chchester, 2002, ISBN 0-471-49691-X

[3] Graham P.: ANSI Common Lisp, Prentice-Hall, New Jersey, 1996 ISBN 0-133-70875-6

[4] Seibel P.: Practical Common Lisp, Apress, Berkeley, 2005, ISBN 9-781-59059-2397

[5] Kiczales G., Riveres J., Bobrow G.: The art of metaobject protocol, MIT Press, London, 1991, ISBN 0-262-61074-4

[6] Newell, A., & Simon, H.A.: GPS: A program that simulates human thought, In H. Billings (Ed.), Lernende automaten (pp. 109-124), R. Oldenbourg: Munchen, 1961

6. Content of the included DVD

Included with the thesis is a DVD containing:

- Directory `src` which contains the framework and subdirectory `examples` which contains examples based on the `cl-ass` framework
- Directory `doc` containing this thesis in pdf, a user manual as a separate pdf, and all the latex source files
- File `readme.txt` containing a small guide on how to load up the framework and how to run examples
- File `license.txt` containing the full license to the `cl-ass` framework